Distributed multiscale computing

Borgdorff, J.

Publication date
2014

2

Theory of distributed multiscale computing[1]

Abstract Inherently complex problems from many scientific disciplines require a multiscale modelling approach. Yet its practical contents remain unclear and inconsistent. Moreover, multiscale models can be very computationally expensive, and may have potential to be executed on distributed infrastructure. In this chapter we propose firm foundations for multiscale modelling and distributed multiscale computing. Useful interaction patterns of multiscale models are made predictable with a submodel execution loop (SEL), four coupling templates, and coupling topology properties. We enhance a high-level and well-defined

[1]The contents of this chapter are based on:
• J. Borgdorff, J.-L. Falcone, E. Lorenz, C. Bona-Casas, B. Chopard, and A. G. Hoekstra. Foundations of distributed multiscale computing: Formalization, specification, and analysis. *Journal of Parallel and Distributed Computing*, 73:465–483, 2013. doi: 10.1016/j.jpdc.2012.12.011

Multiscale Modelling Language (MML) that describes and specifies multiscale models and their computational architecture in a modular way. The architecture is analysed using directed acyclic task graphs, facilitating validity checking, scheduling distributed computing resources, estimating computational costs, and predicting deadlocks. A proposal for how the theoretical results apply to distributed execution is outlined. The methodology is applied to a selected application in nanotechnology, showing its capabilities.

2.1 Introduction

Nature is a complex system that we wish to unravel, understand and sometimes control. Traditionally, science followed the highly successful approach of studying nature as detailed as possible, one part of the puzzle at a time. Extensive data and knowledge have been accordingly accumulated on all spatiotemporal scales, separately. Now we have started to put the pieces back together by studying natural processes holistically as complex multiscale systems. Driven by the availability of abundant amounts of data on all scales, multiscale modelling and simulation of physical, chemical, biomedical, biological and ecological phenomena has become a major activity in science and engineering.

Despite the evident success and relevance of multiscale modelling in many areas of science such as biology and physiology [41, 49, 110, 125–127], material science [29, 50], chemistry [90, 113, 148], and applied mathematics [56], there is little attention to generic multiscale modelling paradigms [67], and related methods of multiscale computing [41, 44]. Moreover, many multiscale models are so computationally expensive that advanced computing capabilities are required, but few initiatives take advantage of the multiscale character of the models to help in this matter [41]. In part, this is because there is no single formal background for multiscale modelling that might help with this [151]. The need for advanced multiscale computing capabilities is expressed by the MAPPER project, representing five different scientific communities facing the need for distributed computing for multiscale models [144]. The main argument for using distributed architecture is that the computational requirements of the single scale models that make up the multiscale model are very heterogeneous,

calling for distinct computing resources. As we will argue, our multiscale modelling paradigm naturally maps to a distributed computing ecosystem, resulting in what we call Distributed Multiscale Computing (DMC).

This idea builds upon the earlier COAST project [143]. That project resulted in a theory of Complex Automata (CxA) [36, 67–70], where several single scale cellular automata that are coupled form a multiscale model. The exact computational architecture of the CxA model can be specified using a Multiscale Modelling Language (MML) [47]. The CxA theory was accompanied by a practical counterpart, a computing environment, first called the Distributed Space Time Coupling Library (DSCL) but later renamed to the MUltiScale Coupling Library and Environment (MUSCLE) [64, 65].

Although CxA and MML only cover coupled single scale cellular automata, here we propose that both concepts can be generalised to cover coupled single scale models. Similarly, MUSCLE is capable of coupling any type of single scale model in a multitude of programming languages, rather than only cellular automata.

Furthermore, for doing distributed multiscale computing MUSCLE already works on self-maintained clusters [65] and it is our goal to extend it for computing on heterogeneous grid environments. In order to achieve this, the computational and communicational requirements of given multiscale models must be adequately predicted and scheduled. If the computation of a multiscale model can be represented by a task graph, by partitioning it into indivisible tasks, quite some research has shown how it can be scheduled on a given set of distributed computing resources [12, 33, 45, 87].

In Section 2.2 we will lay the foundations for distributed multiscale computing by generalising from CxA theory to a formal and comprehensive multiscale modelling theory. The aim of this theory is to be able to define what scales are and how they can be used in multiscale modelling, as well as indicate which interaction patterns are possible in multiscale models. Given these firm modelling foundations, a multiscale model and its computational architecture can be exactly specified with MML, as shown in Section 2.3. This specification can be used for analysis of runtime properties of a multiscale model implementation, and as a guideline for actually executing the model. In Section 2.4 we propose a method to automatically convert an MML specification to a task graph. This task graph serves as an analytical tool to facilitate scheduling decisions on distributed computing resources or as an input to workflow systems. With these tools it is feasible to set up a distributed execution system, using MUSCLE as a coupling library and low-level runtime environment. This approach

is sketched in Section 2.5; However, the practicalities and difficulties of distributed execution of multiscale models are outlined in later chapters.

In Section 2.6 the concepts in this paper will be illustrated by a selected scientific application: a model of the formation of clay-polymer nanocomposite materials [132].

2.1.1 Related work

A number of methodological papers on multiscale modelling exist, each generalising multiscale modelling concepts known so far from the perspective of their respective disciplines, physics [43, 44] and chemical process engineering [76, 113]. They draw from multiscale methods applied to applications so far but do not rigorously define the concepts they use or combine the modelling methodology with concepts useful for implementation. Likewise, Dada and Mendes [41] evaluate the current state of multiscale methodologies and software solutions for multiscale modelling in systems biology and conclude that an all-encompassing solution does not yet exist.

On the other hand, a great number of multiscale concepts, so far loosely described, have been formalised by Yang and Marquardt [151], who define multiscale terms on a conceptual basis rather than an application-driven one. Unfortunately, a fundamental part of their theory considers only spatial scales, which is reflected in their way of representing a hierarchy of submodels based on scale. The formalisation in Section 2.2 offers an alternative to defining multiscale models that considers spatial and temporal scales. One of the achievements of their specification is that they associate it with a machine-readable format in the form of an ontology.

The frameworks classification by Ingram et al. [76] distinguishes different types of couplings between pairs of single scale models. Although this classification shows properties of different frameworks, it does not show why these properties are present. However, by formalising what single scale models are and how they are coupled in Section 2.2, the classification follows from the multiscale properties of a model.

A notable multiscale method, the Hierarchical Multiscale Method (HMM) [42] consists for a large part of strategies to decompose a phenomenon to a multiscale phenomenon. It gives guidelines for when to split certain scales, what methods may be appropriate to certain types of decomposition. Indeed, these strategies are complementary to the methodology proposed in this paper, and mostly adds to Section 2.2.3. Rather, this paper adds to HMM in terms of theoretical scale, and explicit high-level submodel coupling, specification, analysis and distributed computing.

Although MML is a description language of the multiscale domain of discourse, it is not formalised as an ontology to avoid introducing additional terminology. Other languages that describe how components of a program are coupled exist, such as several Architecture Description Languages (ADL's) [5, 9, 55], or the Common Component Architecture (CCA) [7, 8]. Even though both of these architecture descriptions form a respectable basis that influences MML, unfortunately neither describe multiscale properties, which do offer additional insight in multiscale model coupling. Given the additional detail in formalisation since previous work on MML by Falcone et al. [47], we see opportunity to more precisely define MML elements, making them suitable for analysis.

As a general coupling library MUSCLE has alternatives, but as a general multiscale coupling library it does not have an equivalent. Coupling libraries include the open source problem solving environment Cactus[58] CCA-based Ccaffeine [7], the mesh-based MpCCI [78], and earth system modelling frameworks Prism [145] or BFG2 [11], none of which support multiscale models explicitly or directly.

To analyse a distributed execution multiple tools exist besides the task graph, including Petri nets [117, 147] and process calculi [66, 106]. Depending on future needs, the task graph could also be converted to a Petri net, although it is more verbose and tedious in use. The same limitation holds for process calculi, where the latter is also less flexible. Scientific workflows could also be generated from a task graph, to make use of the multitude of workflow software that already exists [15].

2.2 Multiscale modelling formalisation

To make sense of nature's complexity and to do so in a uniform, rigorous, and general way is a difficult task. Multiscale modellers may approach this complexity by functionally decomposing a problem into a set of single scale models that exchange information across the scales, at the same time taking advantage of data available for those scales. Especially when single scale models represent sufficiently different scales, this approach can simplify a problem and strongly reduce the computational cost of a multiscale model. This approach would benefit greatly from firm foundations that make full use of the multiscale character of the system under study. By determining the multiscale character of a model these foundations may offer directions for specific multiscale methods and establish the runtime behaviour of the model.

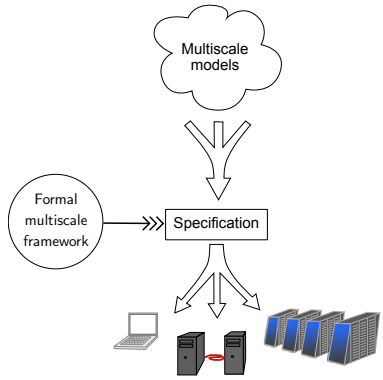


Figure 2.1: An overview what our aim with Distributed Multiscale Computing is: taking multiscale models, defining a formal background to specify them, and running them on heterogeneous infrastructure assisted by their specifications.

Table 2.1: Symbols used in this paper with their type and meaning. Regular and point scale formation have a shorthand notation but represent a special case of a scale specification. The type E is an abstract type for events and type F is an abstract type for submodel states. The last three lines are not mathematical functions but procedures that may have side-effects outside their scope.

Symbol	Type	Meaning
τ, ξ	$\mathbb{R}^+$	characteristic temporal and spatial scale
δ, Δ	$\mathbb{R}^+$	minimum and maximum scale granularity
ω, Ω	$\mathbb{R}^+$	minimum and maximum scale total size
$S(\delta, \Delta, \omega, \Omega)$	$\mathbb{R}^4 \to \text{Bool}$	scale specification (Def. 4)
$S(\Delta, \Omega)$	shorthand for $\mathbb{R}^4 \to \text{Bool}$	regular scale specification (Def. 4)
$S(\Omega)$	shorthand for $\mathbb{R}^4 \to \text{Bool}$	point scale specification (Def. 4)
e, $t(e)$	$E, E \to \mathbb{R}$	event and time of that event
ϑ, ϑ_i	$\mathscr{P}(E)$	time series and future time series
$\mathbf{f}_{\text{init}}$	$\mathbb{R} \to F \times \mathscr{P}(E)$	state initialisation SEL operator (Alg. 1)
S, **B**	$F \times E \times \mathscr{P}(E) \to F \times \mathscr{P}(E)$	solving step and boundary conditions SEL operators (Alg. 1)
$\mathbf{O}_\mathbf{i}$, $\mathbf{O}_\mathbf{f}$	$F \times \mathbb{R} \times \mathbb{R} \to \emptyset$	intermediate and final observation SEL operators (Alg. 1)

'A *multiscale model* is,' to quote Ingram et al. [76], 'a composite mathematical model formed from two or more submodels that describe phenomena at different scales. In this context, we define a *submodel* [...] as a component model that describes only one scale of the system.' According to this definition, a multiscale model is functionally decomposed into submodels describing phenomena at different scales. How to decompose a model into a multiscale model is not specified and as the underlying terms are not well-defined this remains vague. Thus, to adequately work with the terms used in the definition, each of those terms will have to be clearly defined. This includes the terms scale, phenomenon, domain, scale specification and separation ('different scales'), single scale model, multiscale model, submodel and finally the 'composite mathematical model'. The symbols that are introduced in these definitions and in the text are listed in Table 2.1.

2.2.1 Process

In this contribution we restrict our attention to models of physical processes that are bounded in time and space; in short, natural processes. Studying a natural process means that temporal and spatial coordinates can be assigned to it relative to the natural world. Note that this does not include man-made processes. Also note that by taking a more abstract notion of coordinates, other types of processes could also fit into the methodology, but such issues are not explored in this paper.

In the following definitions, we will take the terms natural, process, and observation as primitive terms, and they will not be further defined. Furthermore, we assume that one process can be a part of another. The next term, scale, is defined here on an abstract level and not yet quantified.

Definition 1. *A* scale *is an order of magnitude along a coordinate.*

Temporal and spatial scales are commonly used, but a scale could also be assigned along an abstract dimension, such as a fractal dimension or the number of elements in a set. For natural processes, however, at least temporal and spatial scales are considered.

To quantify the intuition of a scale, the notion of a characteristic scale was conceived [68, 76]. A characteristic scale refers to a process assigns a single number to a scale that can be used as a variable in some models. For instance in reaction-diffusion equations this would be a characteristic temporal scale for diffusion, determined by

the diffusion coefficient, and a characteristic temporal scale for reactions determined by the reaction coefficient [31]. Although in this example the characteristic scale has a well-defined meaning as a model parameter, it will not be defined further here.

The characteristic scale is not set in the process, rather, it is a modelling choice. Even though the characteristic scale may serve intuition, it is difficult to determine how to relate several characteristic scales, or, how far a single scale reaches. Is the characteristic spatial scale of a walking human of 2 meter fundamentally larger than the characteristic spatial scale of the moving legs, of for instance 1 meter? And muscles in the foot, of 2 decimetre? On the other hand, the same human body can be viewed as a single object with some properties or as a collection of cells. In both views the total size is the same, but the scale is intuitively different, even though the body itself does not change during this observation. The issue here is that it is hard to justify that a natural process itself acts only on a certain scale, as opposed to our observation of that process. This is precisely where the term phenomenon is useful: it describes dynamics over an object in the same manner as the term process, but it includes the notion of observation. It is then our assumption that most observations can in some way be discretised, thereby limited to a certain granularity, determining how far a scale reaches.

Definition 2. *A* phenomenon *is a finite amount of data describing a natural process on a given set of scales.*

So a phenomenon is a discretisation of a process, since it is finite. Likewise, a domain is the discretisation of the object of the phenomenon at a certain time. As such, it encompasses all but the temporal dimension of the phenomenon.

Definition 3. *Data of a phenomenon that describes a process at a given time point is a* domain.

A domain D' is a subdomain *of domain D iff they have the same time point and the process that D' describes is part of the process that D describes.*

In contrast to processes, phenomena and domains can be specified by a modeller based on factors such as available data, numerical error, or computational complexity. Concretely, the discretisation may for example depend on microscope resolution, measurement precision or sizes of objects within the phenomenon. The collection of scale specifications is thereby a modelling choice that will be useful for analytic study of the multiscale model, as they are more precise than the characteristic scale.

The following paragraphs will be devoted to clarifying what the collection of scale specifications involves.

We assume that the observed phenomenon has a certain size and a discrete granularity, given the data known about it. Thus, a scale specification should include at least a granularity and a total size, forming the minimum and maximum of that scale. Because the granularity may fluctuate within a single phenomenon, a minimum and maximum may be given to the granularity of a scale specification as well, and likewise for the total size. By providing this range of granularity and total size, the bounds of single scale models become specified. This specification will form the basis for interaction patterns between single scale models in Section 2.2.6.

For example, on a sample of artery tissue of $1.5 \times 1.5 \times 1.5$ mm data was gathered on the lumen size, cell type distribution, and cell sizes. Specifications of the spatial scale of this tissue would have a total size as size of the sample and a variable granularity based on the diversity of cell sizes. If measurements taken are fundamentally more precise than the size of the smallest cell, this can be reflected by taking a very fine granularity. A temporal scale can not be determined in this example, since data only exists of one time point and most importantly only an *object* of study was identified, not a *process*. On the other hand, the spatial scales can be fully specified based on the data available.

For some domains, for instance a regular Cartesian grid, it is not necessary to consider a minimum and maximum granularity; rather, such a grid has a regular scale, with the same step size for each grid point. More extreme, a single value could be seen as a grid with a total size equal to its step size, giving a point scale.

Definition 4. $(\delta, \Delta, \omega, \Omega) \in \mathbb{R}^4$ *is a scale specification, denoted by* $S(\delta, \Delta, \omega, \Omega) = \mathit{True}$ *or by* $S(\delta, \Delta, \omega, \Omega)$*, iff*

$$0 < \delta \leq \Delta \leq \Omega \text{ and } \delta \leq \omega \leq \Omega.$$

A scale specification $S(\delta, \Delta, \omega, \Omega)$ *is* regular *iff* $\delta = \Delta$ *and* $\omega = \Omega$*; its shorthand notation is* $S(\Delta, \Omega)$*. A regular scale specification* $S(\Delta, \Omega)$ *is* point *iff* $\Delta = \Omega$*; its shorthand notation is* $S(\Omega)$*.*

A scale specification $S(\delta, \Delta, \omega, \Omega)$ quantifies a scale of a phenomenon by giving it specific bounds along a coordinate. Here δ and Δ are the minimum and maximum observed granularity, respectively, and ω and Ω the minimum and maximum total

observed size, respectively.

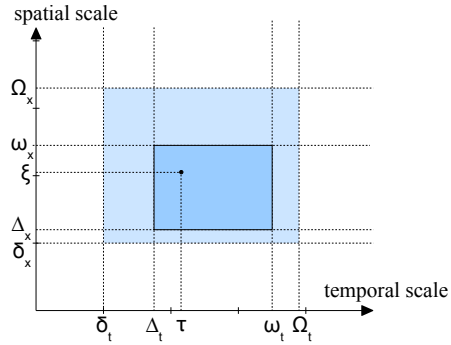


Figure 2.2: A scale map with a spatial and temporal axis indicating the scale specifications and characteristic scales of the phenomenon. The area within the scales of the phenomenon is coloured; the area between minimum and maximum granularity and total size is a lighter colour. The characteristic spatial scale is denoted by ξ and the characteristic temporal scale by τ.

Notably, a temporal scale specification is denoted as $S(\delta_t, \Delta_t, \omega_t, \Omega_t)$ and a spatial scale specification as $S(\delta_x, \Delta_x, \omega_x, \Omega_x)$ as is shown in Figure 2.2. In the rest of the paper, scales are implicitly described by scale specifications.

2.2.2 Scale separation

An important part of multiscale modelling is how the scales of different phenomena relate to each other. As will become clear later this relative scale also affects the structure of the multiscale model and its computation.

Three types of relations will be defined: scale overlap, contiguous scales, or scale separation, each illustrated in Figure 2.3. With these three types of scale relations, it is possible to do meaningful multiscale modelling. Scale separation, for instance, is actively exploited with methods such as scale-splitting [31] or the heterogeneous multiscale methods (HMM) [44].

Definition 5. *Given scale specifications* $s = (\delta, \Delta, \omega, \Omega)$, $s' = (\delta', \Delta', \omega', \Omega')$, *for convenience with* $\Omega > \Omega'$ *or if* $\Omega = \Omega'$ *then* $\Delta \geq \Delta'$*:*

- s *and* s' *are* overlapping *iff* $\Delta < \omega'$ *and* $\Delta' < \omega$*;*
- s *and* s' *are* separated *iff* $\Omega' < \delta$*; and*

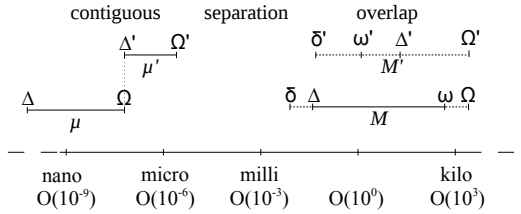


Figure 2.3: Contiguous scales, scale separation, and scale overlap according to the scale specifications of given submodels μ, μ', M, and M', plotted on an arbitrary logarithmic SI scale.

- *s and s' are* contiguous *iff* $\Delta' \leq \delta \leq \Omega' \leq \Delta$.

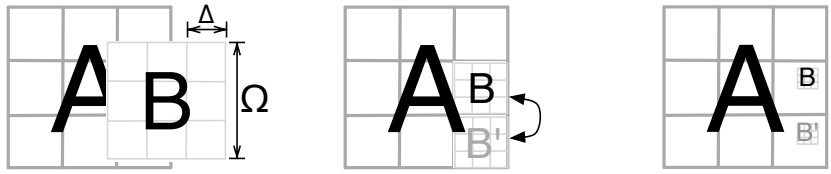
The above definition of scale separation is more strict than the word different, which means non-equal.

Theorem 1. *Two regular scales are either overlapping, separated, or contiguous.*

Proof. Take two regular scales $S(\Delta,\Omega)$, $S'(\Delta',\Omega')$, for convenience with $\Omega > \Omega'$ or if $\Omega = \Omega'$ then $\Delta \geq \Delta'$. Because the scales are regular, the condition for being overlapping simplifies to $\Delta < \Omega'$ and $\Delta' < \Omega$, for being separated to $\Delta > \Omega'$, and being contiguous to $\Delta = \Omega'$. These conditions are clearly mutually exclusive.

To show that they are all inclusive, $\Delta < \Omega'$ must imply $\Delta' < \Omega$. When $\Omega > \Omega'$, combining it with the scale specifications condition that $\Omega' \geq \Delta'$, it follows that $\Omega > \Delta'$. When $\Omega = \Omega'$, using the assumption $\Delta \geq \Delta'$ and the premise $\Omega' > \Delta$, this implies $\Omega > \Delta'$. Thus, the three conditions are mutually exclusive and all inclusive for regular scales. □

A motivation for defining scale separation as above can be found by taking two interacting phenomena A and B of a certain granularity and total size, depicted in Figure 2.4. For example, let them be phenomena acting on living tissue, both represented by a rectangular grid. If the grid cells of A are smaller than the total size of B and the other way around, then for them to interact correctly they will need to exchange information about multiple grid cells. In doing so, they resolve at least part of each others domain. In this case, A and B have spatial scale overlap. When submodels of A and B are treated as black boxes, unaware of each others structure, they will need to exchange information about the entire part of the domain where they overlap at once.



(a) Scale overlap **(b)** Scale contiguity **(c)** Scale separation

Figure 2.4: Two phenomena A and B, both having a rectangular grid as a domain, with different scale relations. Phenomenon B' is of the same type as B, and in Fig. 2.4b it might interact with B, while in Fig. 2.4c it would not. The granularity Δ and total size Ω of B are indicated in Fig. 2.4a.

On the other hand, if one grid cell of A is exactly the total size of B, and B represents exactly one grid cell of A, they have contiguous scales and B will only need information from the one cell in A it represents, and possibly its neighbours. This behaviour is more local, but a submodel of B might need to interact with submodels in neighbouring cells.

When the total size of B is strictly smaller than a grid cell of A, scale separation occurs. It means that B interacts with few grid cells of A, but also that to fully resolve one cell of A, multiple instances of B might be necessary. Since the scale of A is not fine enough to fully resolve B, B might be treated as a canonical example for a single grid cell of A, using its results throughout the cell and not only where B itself is located. This final scenario is used to calculate multiscale speedup [69].

2.2.3 Multiscale model and single scale models

Multiscale modelling takes advantage of data available at distinct scales by modelling interaction between those scales, accordingly managing the complexity of phenomena involved. Practically, first a multiscale phenomenon is functionally decomposed into single scale phenomena, which will then form the basis for several coupled submodels. Having coupled submodels instead of a monolithic multiscale model has the additional advantage of aiding modular development, which in turn benefits implementation [129]. At this stage, however, submodel implementation is not considered, only the model formulation. Given Definition 2, three types of functional decomposition are relevant: decomposition by process, finite observation (or discretisation), and scale.

First, a process consists of dynamics over an object within a time interval. If

any of these terms differ, a different submodel could be used. In practice, the same submodel can be used if the dynamics are sufficiently similar. For example, in two human cells similar processes might occur and those might be represented by the same type of submodel. If the cells have differentiating properties such as cell type or contents, however, a different submodel could be used. Also, if the dynamics of an object change over time, such as when the division of one cell starts, then a different submodel might be used for that.

Second, the discretisation of the phenomenon may be a reason for functionally decomposing a phenomenon. For instance a grid-based model and a agent-based model might both represent the same process, but they allow different aspects of a process to be modelled.

Third, decomposing by scale, also called scale splitting, is possible if several phenomena are taking place on different scales, especially if they have scale separation. For instance, one phenomenon might be changing intrinsically faster than another. A spatial example is suspension flow, which behaves like a fluid on a coarse scale but where the viscosity is determined on a fine scale.

Except scale splitting, none of these functional decomposition methods have a particular dependency on multiscale modelling and arguments for them can be found in components-oriented literature [8, 9, 58]. Nevertheless, we consider it good practice to do functional decomposition and it will help while doing distributed multiscale computing.

Once a multiscale model is decomposed into submodels, each of these submodels should be single scale models. The term single scale model has the intuitive notion of a scale that spans only few orders of magnitude and on which scale splitting is no longer useful to apply. An alternative sense of the term single scale submodel is that the multiscale model additionally contains submodels with different scales.

Finally, when considering model implementation, practical issues may make decomposing a submodel implementation necessary. For instance, take a submodel implementation with computational requirements that scale with the size of its domain, and suppose the submodel turns out to be computationally too expensive to run on a single machine. In order to reduce the computational time, domain decomposition may be performed to split the submodel into multiple instances, each computing only part of the total domain. Although domain decomposition in this case is inspired by practical considerations, generally the submodel on which domain decomposition is

performed also needs to change. This allows it to correctly and efficiently interact between different parts of the domain.

Apart from single scale models, a multiscale model also consists of the interactions between its submodels, indeed, without them a multiscale model would not add anything beyond being a collection. Identifying which phenomena significantly interact is highly domain-specific and is up to the modeller to decide.

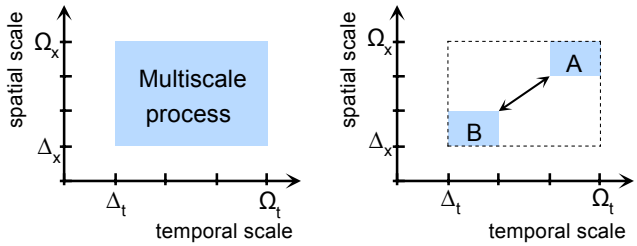
2.2.4 Scale separation map

A way to assess the scales of the phenomena involved in a multiscale phenomenon is to draw its scale separation map (SSM). It is actively used in Complex Automata theory [69] but it can be extended to multiscale modelling in general, as a more basic version of the scale map by Ingram and Cameron [75] shows. Besides helping to gain intuition about the phenomena and their scales, the SSM also communicates the structure of the multiscale model to others.

Concretely, the SSM is a log-log plot of the scales of involved phenomena, with the temporal scale on the horizontal axis and the spatial scale on the vertical axis; a simple example with one phenomenon is shown in Figure 2.2. Once the scales of the phenomena are visualised with an SSM, interactions between phenomena can be indicated by drawing directed edges between their scales. Edges may be labeled with the type of interaction that is involved. For the sake of clarity, phenomena may be removed from the map or merged if they cause confusion or cluttering. During later phases of modelling it may become apparent that the SSM is over- or underspecified and it may be redrawn.

An example of an SSM of a nano materials model [132] is shown in Figure 2.5, with a macro phenomenon of the materials processes on a tangible macro scale and a micro phenomenon of molecular dynamics of the material. These two phenomena have scale separation in both their temporal scale (fast quantum mechanics dynamics versus slower molecular dynamics) and their spatial scale (molecules versus fibres).

Another example, a model of In-stent Restenosis (ISR) as seen in figure 2.6, does not exhibit spatial scale separation: the smooth muscle cell proliferation and blood flow are observed at the same spatial scale. There is temporal scale separation, as blood flow spans a second of a heart beat and a cell cycle lasts a day. The phenomena leading up to ISR, notably the stent placement, are omitted from the map to prevent cluttering.



(a) Multiscale phenomenon **(b)** Single scale phenomena

Figure 2.5: An example SSM of a classical macro-micro model, showing on the axes spatial and temporal scale, with granularity Δ_x, Δ_t; and total size Ω_x, Ω_t respectively. In 2.5a the multiscale model without separating the scales, in 2.5b macro single scale model A and micro single scale model B.

2.2.5 Submodel Execution Loop

In the SSM, edges are drawn to indicate an interaction between phenomena, without stating when information between submodels should be exchanged. To evaluate when information should be exchanged, we will define a few interaction patterns that may be used; and to define these, we will first need to formalise execution patterns of individual submodels. Both submodels and later their implementation should follow these execution patterns.

Of course, a submodel A should only exchange information to submodel B if a state change in A occurs or B triggers a state change in A. The state change itself can be modelled by an *event* $e \in E$, with a model time $t(e) \in \mathbb{R}$ associated to it. In contrast with the definition by Lamport [88] these events are associated with a model time and not with an execution time. The events are then aggregated into a time series that captures all dynamics of a submodel. By relating the time series to the temporal scale specification $S(\delta_t, \Delta_t, \omega_t, \Omega_t)$ of the underlying phenomenon, δ_t prescribes the minimum time step for events to occur.

Definition 6. *A* time series $\vartheta = \{e_0, \ldots, e_n\}$ *is a finite well-ordered set of events with* $t(e_i) < t(e_{i+1})$ *for all* $0 \leq i < n$. *A future time series* ϑ_i *contains events occurring after event* e_i, *so* $\vartheta_i = \{e \in \vartheta \mid t(e_i) < t(e)\}$. *Modifications to* ϑ_i *affect* ϑ *and vice versa.*

Time series ϑ *adheres to temporal scale specification* $S(\delta_t, \Delta_t, \omega_t, \Omega_t)$ *iff* $\delta_t \leq t(e_{i+1}) - t(e_i) \leq \Delta_t$ *for all* $0 \leq i < n$ *and* $\omega_t \leq t(e_n) - t(e_0) \leq \Omega_t$. *A time series adhering to a regular scale specification is regular; one adhering to a point scale specification is a point*

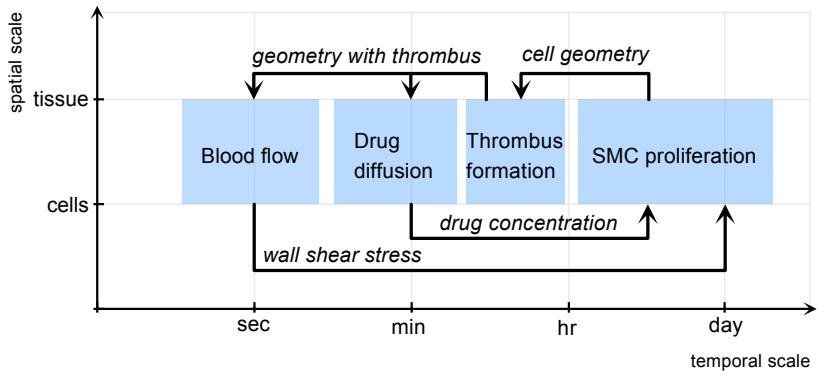


Figure 2.6: The SSM of the ISR model, containing four submodels: blood flow, drug diffusion, thrombus formation and smooth muscle cell proliferation. All submodels act at the same spatial scale $S(1\ \mu\text{m}, 1.5\ \text{mm})$ and at a different temporal scale, from $S(100\ \mu\text{s}, 2\ \text{s})$ (blood flow) to $S(1.5\ \text{hr}, 30\ \text{days})$ (smooth muscle cells).

time series.

Event e_0 and e_n in the definition above are used to determine what the initial and final state of the submodel is, respectively.

Corollary 1. *If a time series ϑ adheres to a scale specification, then $|\vartheta| \geq 2$. A point time series has $|\vartheta| = 2$.*

Because a time series indicates exactly when a state change is observed, it forms a natural guide for executing a submodel. A submodel initialises its state, including boundary conditions, at model time t_0, after which it observes its own initial state. For each subsequent event, the submodel is solved up to that event and the boundary conditions are updated, after which, again, the intermediate state is observed. During updating the state an event might cause changes in future events, hence the future time series are also updated. The time series should then still adhere to the temporal scale of the submodel. When the last event has been processed a final observation of the state is made, thereby concluding the time series and ending the submodel.

The above description of submodel flow can be formalised in terms of a Submodel Execution Loop (SEL).

Here f is the current state of the model, ϑ the time series, i the current iteration, e_i the current event, and t_0 the starting time of the submodel. Five operators are

Algorithm 1: General submodel execution loop

Input: Starting time t_0

$i \leftarrow 0$
$f, \vartheta \leftarrow \mathbf{f}_{\text{init}}(t_0)$
while $|\vartheta_i| > 0$ **do**
  $\mathbf{O_i}(f, t(e_i), t(e_{i+1}))$
  $i \leftarrow i + 1$
  $f, \vartheta_i \leftarrow \mathbf{S}(f, e_i, \vartheta_i)$
  $f, \vartheta_i \leftarrow \mathbf{B}(f, e_i, \vartheta_i)$
end
$\mathbf{O_f}(f, t(e_i))$

used in the SEL: $\mathbf{f}_{\text{init}}$ for initialisation of the state and the boundary; **S** for solving one modelled step or processing up to the time point of the current event; **B** for re-establishing the boundary conditions; and $\mathbf{O_i}$ and $\mathbf{O_f}$ for an observation of the intermediate and final state, respectively. Each of the operators may contain part of a model or procedures. They are not necessarily mathematical functions, rather they may have side-effects such as reading parameters or writing output. Of these operators, $\mathbf{f}_{\text{init}}$, **S**, and **B** modify the state f and future events ϑ_i. Each observation is accompanied by the time point of the last event, and intermediate observations can also take into account the time point of the next event.

Within the SEL, the behaviour of a model is determined by how a modeller specifies the operators, i.e. which actions an operator should perform to reflect the specifics of the phenomenon. Also the way the state or the future events are modified is a modelling choice; the only requirement is that the time series still adheres to the temporal scale specification. However, the execution order of the operators is fixed. This should be reflected both in the submodel itself and in its implementation.

The SEL listed in Algorithm 1 can accommodate different types of models. Event-driven modelling is quite obviously possible, by using the time series as an event queue. Time-driven modelling is also possible, by using a regular time series. Implicit functions, without a time step, can be modelled in several ways, depending on the underlying phenomenon and the flexibility and realism of the function. If the phenomenon has a point time series then there is only one event after e_0 that needs to be solved, so the **S** operator can just contain that function. More specific types of

models also may fit, for example agent based modelling or cellular automata [70]. This wide range of possible model types will allow most multiscale modellers to make use of the SEL formalism.

Each of these modelling types may use an adapted SEL, as long as the order of the operators remains the same as in Algorithm 1. For a time-driven submodel that does not explicitly consider events and that has a regular temporal scale, Algorithm 2 can be used. Since it has a regular time scale and does not need events, the time series will not be changed during the run and can be abstracted away. The SEL operators **S** and **B** could then be specialised to disregard the time series; in the algorithm these versions are denoted $\mathbf{S}'$ and $\mathbf{B}'$. For example, a submodel computing a new state each discrete time step, based only on its previous state, could use this SEL.

Algorithm 2: SEL of a time-driven submodel

Input: Starting time t_0 and temporal scale $S(\Delta_t, \Omega_t)$

$t \leftarrow t_0$
$f \leftarrow \mathbf{f}_{\text{init}}(t)$
while $t - t_0 < \Omega_t$ **do**
  $\mathbf{O}_{\text{i}}(f, t, t + \Delta_t)$
  $t \leftarrow t + \Delta_t$
  $f \leftarrow \mathbf{S}'(f, t)$
  $f \leftarrow \mathbf{B}'(f, t)$
end
$\mathbf{O}_{\text{f}}(f, t)$

A submodel that consists of single function with point scale $S(\Omega_t)$ can for largely be computed in **S**; other operators can then be left almost empty. As shown in Algorithm 3, the time series can again be abstracted away, and having only two events, even the loop can be removed. This SEL could be used by a submodel that computes a state only once or does not generate meaningful intermediate results.

The SEL can also be rewritten to that of a cellular automaton, as used in CxA theory [67]. Generally, as long as the execution order of the operators stays the same it can be rewritten to a range of different forms if that facilitates comprehension or implementation.

Algorithm 3: SEL of a single function

Input: Starting time t_0 and point temporal scale $S(\Omega_t)$

$f \leftarrow \mathbf{f}_{\text{init}}(t_0)$
$\mathbf{O}_\text{i}(f, t_0, t_0 + \Omega_t)$
$t \leftarrow t_0 + \Omega_t$
$f \leftarrow \mathbf{S}'(f, t)$
$f \leftarrow \mathbf{B}'(f, t)$
$\mathbf{O}_\text{f}(f, t)$

2.2.6 Coupling templates

As mentioned, the interaction between submodels is essential to create a multiscale model from a collection of single scale models. Indeed, this coupling of submodels may be the primary object of interest in the model and may be as computationally expensive as the submodels themselves. In contrast to scale bridging techniques used in a coupling, which may differ from one model to the next, we argue that the frequency of interaction between single scale models exhibits regularity. Analysing patterns in the frequency of interaction gives insights into the dynamics of the multiscale model itself as well as its runtime behaviour.

Coupling templates, defined as a data transfer between the SEL operators of two submodel instances, are a means to specify these patterns. To limit the number of possible data flow patterns, and thus coupling templates, the operators $\mathbf{O}_\text{i}$ and $\mathbf{O}_\text{f}$ are only allowed to send data while the operators $\mathbf{f}_{\text{init}}$, **B** and **S** may only receive data. This restriction makes coupling templates inherently unidirectional, as they are defined only between a pair of operators, transferring data from one to the other. There are six possible combinations of operators, each with its own interpretation, four of which are listed in Table 2.2.

The **B** operator is not listed separately among the coupling templates in Table 2.2, having a similar role as **S** concerning the frequency of interaction. Instead, their role is distinguished by the concepts of multidomain and single domain coupling.

Definition 7. *A coupling between two submodels A and B with domains D and D′, respectively, is* single domain (sD) *iff D is a subdomain of D′ or D′ is a subdomain of D. Otherwise, the coupling is* multidomain (mD).

The general distinction between boundary operator **B** and solving operator **S** in

Table 2.2: Coupling templates between submodels A and B, listing: its name and operator; what the temporal relation between the operators in A and B is; what the temporal scale relation between A and B is; and a common scenario in which the template is used.

Coupling template		Time	Temporal scale	Scenario
1. interact	$\mathbf{O}_\mathbf{i}^A \to \mathbf{S}^B$	$t_A < t_B$	overlap	reciprocated by the same template
2. call	$\mathbf{O}_\mathbf{i}^A \to \mathbf{f}_{\text{init}}^B$	$t_A \leq t_B$	contiguous/separation	reciprocated by template 3
3. release	$\mathbf{O}_\mathbf{f}^B \to \mathbf{S}^A$	$t_B \leq t_A$	contiguous/separation	B was started by template 2
4. dispatch	$\mathbf{O}_\mathbf{f}^A \to \mathbf{f}_{\text{init}}^B$	$t_A \leq t_B$	any	loosely coupled or stateful

coupling templates is that **B** is used to receive messages in mD couplings while **S** is used in sD couplings. The motivation for this distinction is that boundary conditions generally should not deal with what happens within their own domain, but rather what happens when interacting with another domain.

For example in the ISR model, the blood flow in the lumen has two type of boundaries: the first is the blood vessel wall, which is defined by the smooth muscle cell proliferation; the second is the artificial boundary of the vessel, with a certain amount of blood flowing in and out of the domain. In this case, the artificial boundary is resolved within the blood flow model, whereas the blood vessel wall is resolved as a mD coupling with the smooth muscle cell domain. On the other hand, the drug diffusion submodel in the uses the same domain as the smooth muscle cell proliferation submodel, making it an sD coupling.

The use of coupling templates can be interpreted to a certain extent. First, if two single scale models A and B have temporal scale overlap then they will need to resolve multiple time steps of each other. Within the SEL this is possible by using the $\mathbf{O}_\mathbf{i}$ and **S** (or **B**) operators, so they will *interact* for each event in both submodels. This interaction will only occur after the event has happened, so calling $\mathbf{O}_\mathbf{i}$ will happen at an earlier time than that it is processed by **S**. Conversely, if the single scale models are scale separated and A has a larger temporal scale than B, then there is the possibility that B needs to be resolved completely once per time step of A. Hence, A may *call* B during each time step and when B is finished it may *release* its observations and send them to A. When calling B, A gets the time of $\mathbf{O}_\mathbf{i}$ as starting time, and it must release, according to their contiguous or separated scales, before the next solving step of A. If A is an initialisation or *ab-initio* single scale model and it is finished, it may *dispatch* its state to B, which will continue computation at the time that A left off. Alternatively,

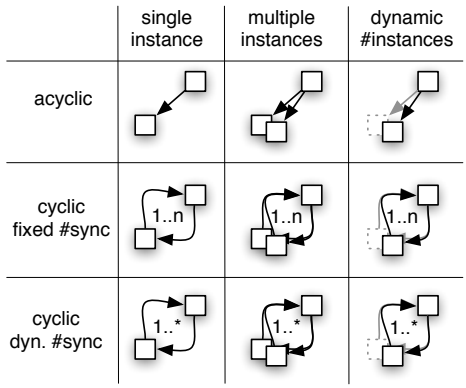


Figure 2.7: Graphical overview of different coupling topology properties, like being cyclic or acyclic, having a fixed or dynamic number of submodel instances, and having a fixed or dynamic number of synchronisation points. Shown here are only examples with two submodels, but this can be generalised to any number of submodels.

A may just dispatch part of its state to a next instance of itself, making *A* stateful.

These interpretations of the coupling templates are not all-encompassing but do in our view provide the main reasons for using them.

In the example of the ISR model, none of the submodels has temporal scale overlap with another, so the interact template is not used. The dispatch template is used when a submodel calculates the initial conditions for the model and sends them to the smooth muscle cell proliferation submodel. This submodel then calls thrombus formation, which in turn dispatches to blood flow and drug diffusion. These submodels then release to smooth muscle cell proliferation, after which the loop continues.

2.2.7 Coupling topology

Coupling templates indicate what data flow may occur between a pair of submodels, but for interpreting a multiscale model its full topology has to be considered. A coupling topology can be defined as a graph representation of a multiscale model, with coupling templates as its edges, weighted by the number of times data is sent over them, and instances of submodels as its nodes. This is similar to a scale separation map, although the latter does not have submodels instances but only submodels, and does not know how often couplings are used. This edge weight, combined with the SEL, will help understand how often a particular part of a model will be run.

A few properties of coupling topologies are of interest when computing or distributing a multiscale model. Three properties are are listed here and visualised in Figure 2.7.

1. Is the coupling topology cyclic or acyclic, or does only parts contain cycles?

2. Are there multiple instances of certain submodels, and if so, can the number of instances be statically determined?

3. Can the number of synchronisation points be statically determined?

If a multiscale model has an acyclic coupling topology, computation becomes quite straightforward: the submodels can then be ordered and executed consecutively. Otherwise, a more complicated execution model has to be used, like dynamic execution or one that predicts which parts of submodels are scheduled when. When the interact coupling template is used and reciprocated, a coupling topology is cyclic, but a cycle can also be formed with any of the other coupling templates. A cycle can include any number of submodels.

The second property, multiplicity, gives an indication for the number of submodel instances that are needed. If there is more than one instance per submodel, these instances need to be addressed separately and a means for forking and joining data might be necessary. If the multiplicity is determined during the runtime of the model, there needs to be a mechanism for the model to spawn new instances of submodels, increasing the complexity of the model and the framework that would run the model.

Having spatial scale separation can be a reason for needing multiple instances of a submodel. Take two submodels A and B with spatial scale separation and A being the larger of the two. The domain of A can then contain a multitude of subdomains modelled by B, needing many instances of B to model them all. If the phenomenon modelled by A is not regular in time or space it might not be predictable how many instances of B are needed and the number of instances needed could even change over time.

Finally, the third property indicates the number of times each submodel will transfer data and do computation. If static, such as when the temporal scales of all submodels are regular, data transfers and submodel execution times can be predicted and thus scheduled. Otherwise, if the model communication is self-organised within the model then the amount of running time is harder to estimate, or if communication is sched-

uled then the scheduler must be informed of the number of communications and calculations that will take place.

The three properties defined here have a large impact on the complexity of software to compute the multiscale model, where a more dynamic property means more complex software.

2.3 Specifying a multiscale model

The previous section introduced scales, submodels and coupling templates and topologies. The Multiscale Modelling Language (MML) is a high-level way to describe and specify these concepts, along with the computational architecture of a multiscale model. [47]. First of all, MML describes the code implementation of submodels and coupling templates as computational elements. At this stage MML only needs an overview of the implementation, like inputs and outputs and the scale that is used. Second, it introduces computational elements to specify how messages will be distributed between submodels and how they will be affected during transit. Finally, code characteristics of computational elements may be specified to guide a runtime environment in how to execute those elements.

MML has multiple forms:

- a graphical format, hereafter denoted gMML and
- a human-writable, machine-readable XML-based format, denoted xMML.

Between them, xMML is more complete and precise while gMML is more human-friendly. First gMML can be created in a graphical editor and form part of the documentation of a multiscale model or serve as input for creating an xMML file. Then xMML serves as input for generating a scientific workflow or a rudimentary code implementation. Most concepts sketched here were already available in the original presentation of MML, but below they are refined and adapted to the terminology of the previous section. Moreover, the concepts here do not depend on Complex Automata, but rather they are extended to the multiscale modelling formalism sketched in the previous section.

2.3.1 Computational elements in MML

The primary computational element in MML is a submodel implementation. Both the submodel and its implementation can be described in MML. A computational element has input ports, on which it receives messages, and output ports, on which it sends messages. Connections between computational elements are made with conduits. A conduit is a unidirectional connection between an output port and an input port and it transfers data in the form of messages. A message contains data, for instance an observation of a submodel state, the time point of the event it is associated with, and the time point of the next message that will be sent over the conduit, if any.

Submodels may define ports based on their SEL operator. This means, that output ports may be defined for operators $\mathbf{O_i}$ and $\mathbf{O_f}$ and input ports for $\mathbf{f_{init}}$, **S** and **B**. A coupling template between the operators of two submodels can then be translated as a conduit between an in- and output port defined for those operators. For predictability and to prevent deadlocks, submodels should send a message each time the operator that a port is defined for is reached.

A multiscale modelling software framework used with MML may be data-driven, allowing each submodel to execute independently until a message from another submodel is needed, which the submodel will await. A multiscale model can then be initialised by starting a subset of submodel instances, and ended by consequence of the final operator of the SEL of a subset of submodel instances.

Submodels so far defined do not have a state once their SEL is finished. A way to explicitly save a state outside the SEL is to have a coupling from the final observation of the submodel instance to the initialisation of the next time that it is called. Since this iteration will only be instantiated if another submodel needs it, it does need separate semantics.

To increase the independence of different submodels, one submodel should not concern itself with sending a data format that is acceptable for the other, nor should the other submodel concern itself with what submodel it is getting data from. Instead, MML has a *conduit filter* or just filter that can be applied to a conduit. The filter is then applied to messages transferred over that conduit to ensure that both submodels handle the data in a way most convenient to themselves and still communicate correctly. Conduit filters should not model phenomena, but rather do a physically correct data transformation that enables each submodel to reason about its own data. The modelling of these filters alone can be a challenging part of multiscale modelling,

for instance when mapping grid-based data to particle based data or doing fine-graining or coarse-graining. Filters are allowed to have a state, so they can do time averaging by aggregating data coming in, possibly altering the associated time point in the process. They are reactive as they may not produce data if no data is sent over the conduit, but not inherently predictable, as they may send multiple or zero messages when receiving a single message. Instead, they are predictable by specification, the ratio of the number of messages sent versus the number of messages received has to be specified in advance. In contrast to submodels, filters may know exactly what kind of data the sending submodel will generate and what kind of data the receiving submodel expects. This allows submodels to stay modular.

Consider for instance two submodels A and B with temporal regular scales $S_A(2\text{ s}, 1\text{ hr})$ and $S_B(1\text{ s}, 1\text{ hr})$ that are coupled on a single domain. They have temporal scale overlap and should use the interact coupling templates $\mathbf{O}_\mathbf{i}^A \to \mathbf{S}^B$ and $\mathbf{O}_\mathbf{i}^B \to \mathbf{S}^A$. However, to allow each submodel to have their own time step and to let the coupling stay correct, a temporal conduit filter should be used on conduits between A and B. Events should be interpolated in the direction from A to B to ensure there are enough messages for B to process; in the other direction, events should be aggregated to ensure that A does not receive too many. Appendix A shows an algorithm for a temporal scale conduit filter that does this, accompanied by a proof of being deadlock-free.

Since the filter applies to only one conduit, it can not do a data transformation for which multiple sources are necessary, nor can it distribute the transformed data to multiple submodels, For these situations a *fan-in* or *fan-out mapper* should be used. A fan-in mapper has one output port and n input ports, which may accept different types of data. Formally, a fan-in mapper is a procedure $m : I^n \to O$, where I is an abstract data type of the input ports that may be further specified per port, and O is the output data type. A mapper is procedure rather than a function, since it may interact with the world. A fan-out mapper has one input port and a n output ports, which may produce different types of data. Again, formally, a fan-in mapper is a procedure $m : I \to O^n$, where I is the input data type, and O is an abstract data type of the output ports that may be further specified per port. Mappers can also have conduits to other mappers, and those conduits may also have conduit filters. For convenience, mappers may perform non-trivial data transformations, reducing the need for filters and thus simplifying the computational architecture. However, combining trivial join and split operations in mappers with filters can in principle

yield any data transformation.

When doing domain decomposition or when the number of submodel instances is dynamic it can be useful to instantiate a number of submodel instances at the same time. However, to maintain their modularity, a submodel should not know how many submodel instances it should communicate with. Otherwise, the submodel implementation would have to change each time the domain is decomposed over a different number of submodel instances. By having an intermediate fan-out mapper between the submodels, submodels only need to know what data to send and not how to divide it among submodel instances. Conversely, those submodel instances should not have to know how the workload or domain is divided among themselves, and just send any data to a fan-in mapper that will take care of that. For the disassembly and assembly to happen correctly, the fan-out mapper could also send a message to the fan-in mapper on how it distributed the workload so that the fan-in mapper can correctly reconstruct the data again. With a fan-out and fan-in mapper it is thus easy to create multiple submodel instances simultaneously and address them throughout the multiscale model.

Finally, each port in MML needs to be connected. To test computational elements in isolation or in arbitrary configurations, ports that do not have a viable counterpart may be connected to a terminal. A source terminal generates data at the time scale of the receiving port, and a sink terminal receives data at that rate. A terminal can be used to read or write data from or to file; to simply discard data; to read from a URL; or to generate data according to an algorithm. By connecting a single submodel to terminals, the response to a range of input values can quickly be explored, and output data can easily be validated.

2.3.2 Graphical MML

Being inspired by UML, gMML is a graphical representation of MML with some UML icons. An example of the gMML of an in-stent restenosis model is shown in Figure 2.8. A submodel instance is shown as a rectangle with a name label inside. Mappers are drawn as a hexagons, with their name labels inside, and a conduit filter as a rounded square, with a name label inside or close to it. If a number of submodels have a single domain coupling, this may be emphasised by drawing a dotted rectangle around the involved submodels, optionally with the name of the domain involved.

Conduits are represented as edges between elements, which may be labeled with

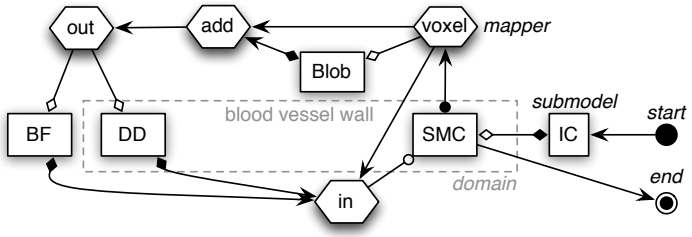


Figure 2.8: The gMML of the ISR3D model, which is described in Chapter 4.4. Labels in italics indicate the different types of computational elements. The different edge heads and tails show the operators that are used, depicted in Figure 2.9.

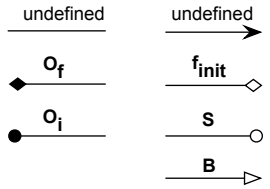


Figure 2.9: The different edge heads and tails corresponding to the SEL operators. Undefined means no SEL operator is associated with the coupling.

the type of data that is sent over it. Based on coupling templates, edge head and tail decoration depends on the SEL operator that a conduit is sending to and from, respectively, as shown in Figure 2.9. Operators $\mathbf{f}_{\text{init}}$ and $\mathbf{O}_\text{f}$ are shown as a diamond; operators $\mathbf{O}_\text{i}$ and $\mathbf{S}$ as a circle, being called within the loop; and operator $\mathbf{B}$ is shown as an empty arrowhead. Edge tails are filled black, edge heads are empty. Since a mapper does not have an SEL, edges from or to it have no tail or a simple arrow head, respectively.

The submodel instances that are initially instantiated have an edge from a filled black circle; submodel instances that determine when the model will finish have an edge to a filled black circle with a black round border. These circle icons correspond to the start and end state of a UML state diagram.

Limitations of gMML include having no way to see the scales involved, and not being very detailed. Its advantage is that it is a visual way of thinking about the computational architecture of a multiscale model, and provides a quick way to compose a multiscale model in a user interface.

2.3.3 XML format of MML

To automatically process the specification of a multiscale model a machine-readable format is needed, which is preferably also human-writeable. The XML specification has the advantage that it is well-known and that there are many software tools available to edit or parse it. The xMML format is specified with an XML Schema, with which a minimal specification can be enforced and xMML elements may be validated. By using namespaces, the format can be extended with XML extensions such as XInclude, XPointer or RDF, adding some flexibility to the language. These specifications combined form the basis for xMML.

The content of xMML is aimed to be both descriptive and prescriptive. It is descriptive in the sense that it describes in detail the submodels, their computational requirements, and the couplings that they have. It is prescriptive in the sense that: during execution submodels should conform to the scales specified; and a port of a submodel should only be used at the SEL operator it is defined for. The descriptive part enables an execution framework to run and schedule a multiscale model by reading its xMML specification. The prescriptive part allows predicting and possibly enforcing when messages will be sent to which computational element. Together, the descriptive and prescriptive sides of xMML allow a correct execution of a multiscale model.

In Listing 2.1 an example of the xMML of a macro-micro model is given, with a two-dimensional macro submodel A and a set B of ten one-dimensional micro submodels. The corresponding gMML is depicted in Figure 2.10. The macro submodel starts executing, then transfers an observation in the form of a grid to a fan-out mapper, which distributes different values to the micro submodels. Afterwards, the reverse direction is executed using a fan-in mapper, after a filter has been applied to reduce the resulting 1D array to a single value. All this is specified within the `<topology>` tag. The combination is a classical case of macro-micro scale separation, using the coupling templates call ($\mathbf{O}_{\mathbf{i}}^{A} \rightarrow \mathbf{f}_{\text{init}}^{\ B}$) and release ($\mathbf{O}_{\mathbf{f}}^{B} \rightarrow \mathbf{S}^{A}$) with a single domain coupling, since the micro submodels use a subdomain of the macro submodel. In general, to make the domain used explicit, an `<instance>` tag may contain a domain attribute.

More detailed specifications are available within the `<definitions>` tag. Here, the scales of the submodels are specified using SI notation or with time notation in terms of minutes, hours, *etc.* If no SI unit is given, the default SI unit is presumed, seconds for time and meters for space. In the example, all scales are regular, except

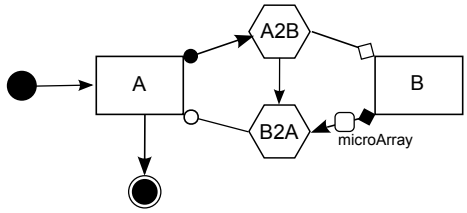


Figure 2.10: The gMML of a simple macro-micro coupling, corresponding to Listing 2.1.

that of the second spatial dimension of the macro submodel, which uses the complete scale specification. In the <ports> tag, inputs and outputs of the submodel to which conduits may attach are defined. An SEL operator is assigned to them to associate conduits with a coupling template, and to predict, together with the temporal scale, how often the port will be used. A data type is also assigned to the ports, to later ensure that both the sender and receiver use the same type of data. With <mapper> tags the ports and function of fan-in and fan-out mappers can be specified; with the <filter> tag a datatype converter or scale reduction or interpolation may be specified; and a <terminal> defines a source or sink terminal. Moreover, a model and submodels may be versioned, described and named, and all implementable computational elements may be given an executable class name.

The root element <model> lists the name of the multiscale model, the version of the current document and the version of xMML that is used.

Listing 2.1: The xMML specification of a macro-micro model (continues on next page).

```
<?xml version="1.0"?>
<model id="MacroMicro" xmlns="http://www.mapper-project.eu/xmml" name="
    Macro-micro model" version="1.0" xmml_version="0.3.3">
  <description>A macro-micro model, having a macro grid representation and
      a micro cell representation.</description>

  <definitions>
    <datatype id="lattice2DDouble" size_estimate="x*y*sizeof(double)"/>
    <datatype id="latticeMapping" size_estimate="x*sizeof(int)"/>
    <datatype id="valueDouble" size_estimate="sizeof(double)"/>
    <datatype id="arrayDouble" size_estimate="x*sizeof(double)"/>
    <filter id="microArray" type="converter"
        datatype_from="arrayDouble" datatype_to="valueDouble"/>
```

```
  <submodel id="Macro" name="2D Macro model" init="yes">
    <timescale delta="1 s" total="1 min"/>
    <spacescale delta="1 mm" total="1 dm"/>
    <spacescale>
      <delta min="0.7 mm" max="1.3 mm"/>
      <total min="0.7 dm" max="1.3 dm"/>
    </spacescale>
    <ports>
      <out id="grid" operator="Oi" datatype="lattice2DDouble"/>
      <in id="gridDiff" operator="S" datatype="lattice2DDouble"/>
    </ports>
  </submodel>

  <submodel id="micro" name="1D Micro model">
    <timescale delta="1E-7" total="1E-5"/>
    <spacescale delta="1E-5" total="1E-3"/>

    <ports>
      <in id="start" operator="finit" datatype="valueDouble"/>
      <out id="diff" operator="Of" datatype="arrayDouble"/>
    </ports>
  </submodel>

  <mapper id="gridDivide" type="fan-out">
    <ports>
      <in id="grid" datatype="lattice2DDouble"/>
      <out id="mapping" datatype="latticeMapping"/>
      <out id="value" datatype="valueDouble"/>
    </ports>
  </mapper>
  <mapper id="gridCombine" type="fan-in">
    <ports>
      <in id="value" datatype="valueDouble"/>
      <in id="mapping" datatype="latticeMapping"/>
      <out id="grid" datatype="lattice2DDouble"/>
    </ports>
  </mapper>
</definitions>

<topology>
  <instance id="A" submodel="Macro"/>
  <instance id="B" submodel="micro" multiplicity="10"/>
```

```
    <instance id="A2B" mapper="gridDivide"/>
  <instance id="B2A" mapper="gridCombine"/>

    <coupling name="grid" from="A.grid" to="A2B.grid"/>
    <coupling name="values" from="A2B.value" to="B.value"/>
    <coupling name="mapping" from="A2B.mapping" to="B2A.mapping"/>
    <coupling name="diffs" from="B.diff" to="B2A.value">
      <apply filter="microArray"/>
    </coupling>
    <coupling name="grid diffs" from="B2A.grid" to="A.gridDiff"/>
  </topology>

</model>
```

In addition to the modelling aspects, the implementation and computational aspects of a multiscale model can be specified using xMML. Above, the size estimate included in the data type, based on the scales of the involved submodels, is used to estimate the size of messages with that data type. Instead of relying on this implied message size, the expected size may also be explicitly listed for each coupling. With this size estimate, submodels instances may be placed closer together in a distributed environment based on the amount of data transferred between them. In submodels, mappers, filters, and terminals an `<implementation>` tag may be added to specify the estimated memory and CPU requirements, as well as stating library or platform dependencies. Combined, the implementation descriptions allow scheduling software to make more informed decisions on distributing submodels, mappers, and conduits.

In addition, the implementation of an element may be included in the XML file, which can be used to generate the appropriate simulation code and executables. This is especially convenient for XML-based descriptions of submodels, such as CellML [93] or SBML [49].

2.4 Multiscale model analysis

Being able to predict the runtime behaviour of a modular program has benefits for scheduling, estimating computational costs, deadlock detection, and overall validity checking. A multiscale model, as we are proposing, is modular and has internal structure through the submodel execution loop, coupling templates, coupling topology, and computational elements. In this section we obtain the coupling topology of a

multiscale model and propose a method to predict the models data flow from its MML specification.

Several formal models for task dependencies and data flow in general exist, for instance task graphs, Petri nets, process algebra, or I/O Automata. In addition, the more informal scientific workflows also have associated tools to predict runtime behaviour of modular software. All these methods importantly have some form of software support, as automation gives a feasible and consistent way to predict runtime behaviour. We will use task graphs, being the simplest for our purposes.

2.4.1 Coupling topology: deduction and implications

The coupling topology of a multiscale model can be deduced from its xMML specification. Firstly, whether a coupling topology is cyclic or acyclic can be determined from this specification in linear time by constructing the graph formed by submodel instances as nodes and couplings as edges.

Secondly, the multiplicity of the submodels is explicitly specified in xMML, and can again be verified in linear time.

Finally, whether the number of synchronisation points is dynamic can be determined by evaluating the temporal scales of the coupled submodel instances. If all submodels with an output port at their intermediate observation have regular scales then the number of synchronisation points will be static (or 'fixed').

The implications of different coupling topologies have the same tendency: a more dynamic setting makes it more difficult to manage and predict the runtime behaviour of a multiscale model. In each case the executing framework must anticipate dynamic changes in the number of computational resources required, and manage communication in a dynamic environment. Moreover, since it is up to the model to decide what the dynamic requirements are, the model must also be adapted to communicate dynamic needs to the execution framework.

2.4.2 Predicting runtime behaviour with a task graph

In this article we analyse the runtime behaviour of multiscale models using task graphs, a formal but concise way of representing task dependencies and data flow between tasks. A task graph is a directed acyclic graph of tasks as nodes and their dependencies or data flows as edges, as for example in Figure 2.12a. It is used primarily

for scheduling on parallel and/or distributed computing resources [87]. A close concept is that of the wait-for graph [103], used for deadlock detection in distributed computing. It can also be seen as a serialised or unfolded graph of the MML description, which may be cyclic. Although task graphs themselves are well-known, converting a problem to a task graph in a nice way is problem-specific. In this section we convert an MML description to a task graph, which has not been done before.

A multiscale model can be subdivided into submodel instances, and those can be subdivided into SEL operators in conjunction with the iteration of the execution loop. These SEL operators are the main nodes of the task graph. Mappers are also explicit nodes in the task graph, conduit filters on the other hand may be implicit as long as they do not aggregate or interpolate over the temporal scale. Conduits are modelled as dependencies and thereby form the edges. Submodel instances, mapper instances, or conduit instances that are initiated multiple times, for instance from the loop of another submodel, are distinguished by an initiation number. Multiple submodel instances that are initiated at once, for instance for purpose of domain decomposition, form a set. Another type of dependency is that of the internal flow of a submodel, from one of its SEL operators to the next; or more broadly, the state of a submodel to the next in stateful submodels. These dependencies are implicit to the submodel flow and are consequently displayed with a dashed edge instead of a solid one. Together, SEL operators and mappers form the nodes of the task graph, and conduits and internal submodel transitions the edges.

A task graph node representing an SEL operator has a label showing that operator. The full format is $instanceName_j[k](o,i)$, where *instanceName* is the name of the computational element, o the SEL operator and i the iteration. Then j indicates the initiation number of a submodel instance that is initiated multiple times. If a set of submodel instances is concerned, an index k is added. This format can be simplified by leaving out any part, as long as a task remains uniquely identifiable.

The task graph of models with a static number of synchronisation points and submodel instances can be statically generated in a linear fashion, as shown in Algorithm 4 and 5. The algorithm depends on the SEL and on the coupling of MML's computational elements and as far as we know is an original contribution. If it were to be dynamically generated, while the model is executed, it would not require significant changes, other than some form of communication with that model. The algorithm starts with a number of submodel instances that are specified to be initial, and keeps

track of active computational elements. These computational elements have to be well-specified, so mappers and submodels work quite straightforward, although they do have to keep track of their timestamps. A submodel additionally has edges to its next operator and if it is stateful, also to its next instance. Mappers need a message on each input, and once a message has been received on that input, subsequent messages are buffered. In a task graph, this can be modelled by sending the message to the next instance of the mapper. For conduit filters, Algorithm 6 must be evaluated to see where it will send its output, and when all required messages are received.

In order to keep Algorithm 4 relatively brief, it does not include the temporal conduit filter. Support for it can be added at lines B and D, by keeping track of how many messages the filter must receive before it sends a message, and how many messages the filter sends when it finally does send. Likewise, the algorithm does not support conduits with multiplicity, which could be added with a simple for loop at line D.

Deadlock detection happens in two places: while building the task graph and afterwards. During building the task graph, a task that has been active before could be referenced again. Since this can only happen if that task already had all dependencies resolved before, there must be a cycle that causes a message to be sent to it twice. The second time deadlock detection becomes active is after building the task graph, when it can find that some elements were still waiting for input but could somehow not receive it.

In Algorithm 4 three lines A, B, and C indicate what part of the task graph may be distributed. At A, a computational element doing a computation at a certain operator may decide to send a number of messages. Deadlock detection is done by the receiving computational element, which will detect that the timestamp is not correct. At B, it has to wait for all messages to arrive before it can do its own computation, which is built in to the MML specification of an element. The deadlock detection at C, however, has to be changed to distribute efficiently, seeing that it is activated only after all operators and submodels were executed as long as they could. At this point, alternative distributed deadlock detection algorithm should be used.

Task graph acyclicity

The task graph algorithm presented indeed gives an acyclic task graph, simply because any cycle will throw a deadlock. Here we show that in most well-defined cases, dead-

Algorithm 4: Generate a task graph, while doing deadlock detection. Variables a and b are potential task graph nodes. Part 1/2.

Input: A full MML specification, with a set of submodel instances Σ
Output: Task graph $G(V, E)$, with vertices V and edges E

```
// initialise the graph;
add src, sink to V;
foreach s ∈ Σ that is marked initial or has no incoming conduits do
    add s(0, f_init) to active;
    add src → s(0, f_init) to E;
end
// activate one node at a time, and calculate outgoing
edges;
while active is not empty do
    a ← remove element from active;
    add a to V;
    next ← nextSet(a);
A   foreach b in next do
        if b has already been active then
            Deadlock: cycle in the task graph detected;
        end
        add a → b to E;
        add b to sleeping;
    end
    if next is empty then add a → sink to E;
    ;
B   while sleeping contains element b that received all required messages do
        move b from sleeping to active;
    end
end
C if sleeping contains elements with other incoming edges than stateful edges then
    Deadlock: elements in sleeping still require incoming edges but there are no
    active senders;
end
```

Continued on the next page as Algorithm 5...

Algorithm 5: Generate a task graph, while doing deadlock detection. Variables a and b are potential task graph nodes. Part 2/2.

```
D define nextSet[a = s_j{i, o}], s is a computational element
    if s is a submodel instance then
        if o = O_f then
            if s is stateful then add s_{j+1}(0, f_init);
            ;
        else if o = B and |ϑ_i| > 0 then
            add s_j(i, O_i);
        else
            o' ← SEL operator after o (f_init → O_i → S → B → O_f);
            i' ← i + 1 if o = O_i, else i;
            add s_j(i', o');
        end
    end
    foreach outgoing conduit of s to operator o' of s' do
        (i', j') ← previous iteration and initiation number used in this conduit
        (default: 0, 0);
        if o' = f_init or s' is a mapper then increment j';
        ;
        if o' = S or o' = B then increment i';
        ;
        add s'_{j'}(i', o');
    end
end
```

lock as caused by a cycle will not present itself. To show this, assume that for the interact, call, and release coupling templates; scale separation, overlap, or contiguity is strictly adhered to. To begin with, we will show when pairs of submodels yield an acyclic graph. Thereafter, we will show when this will also yield a total acyclic task graph. The timelines of the different submodels, determining the eventual structure of the task graph, are shown in Figure 2.11.

For the first part, consider submodels A and B with temporal scales $S(\delta_t, \Delta_t, \omega_t, \Omega_t)$ and $S(\delta'_t, \Delta'_t, \omega'_t, \Omega'_t)$ that have temporal scale separation with $\Omega'_t < \delta_t$, and that they are coupled with the call and release coupling templates. A message sent by A at $\mathbf{O_i}$ at time t, arrives at B at $\mathbf{f}_{\text{init}}$ at time t, a message sent by B at release arrives at time at most $t + \Omega'_t$. By that time, it reaches the next iteration of A, which has time at least $t + \delta_t > t + \Omega'_t$. In terms of task graph nodes this can be expressed as

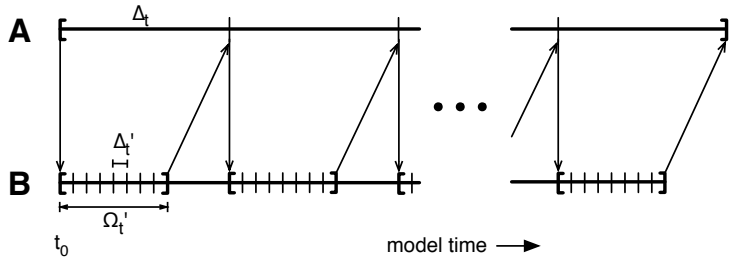
$$A(i, \mathbf{O_i}) \to B(0, \mathbf{f}_{\text{init}}) \rightsquigarrow \cdots \rightsquigarrow B(n_B, \mathbf{O_f}) \to A(i+1, \mathbf{S}),$$

for any $0 \leq i < n_A$, $n_A + 1$ and $n_B + 1$ being the length of the time series of A and B.

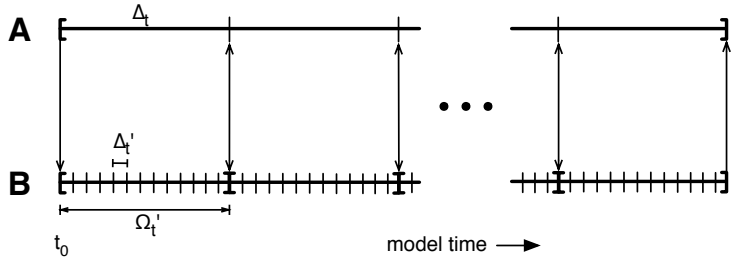
Next, if A and B have contiguous scales instead, specifically $\Delta'_t \leq \delta_t \leq \Omega'_t \leq \Delta_t$, then it is not clear that the time $t^A \in [t + \delta_t, t + \Delta_t]$ for one iteration of A is longer than the time $t^B \in [t + \omega'_t, t + \Omega'_t]$ for B to release. On the contrary, this might create an overshoot of at most $\Omega'_t - \delta_t \leq \Delta_t - \delta_t$. It is up to the modeller to prevent this overshoot; failing to do so can lead to cycles or deadlocks. With regular scales, however, this problem does not arise.

Finally, A and B can have temporal scale overlap, with $\Delta_t < \omega'_t$ and $\Delta'_t < \omega_t$. As mentioned in Section 2.3.1, we only consider this case under the condition that the granularity of the receiving submodel is fixed, so if the interact coupling template $\mathbf{O}_\mathbf{i}^A \to \mathbf{S}^B$ is used then $\delta'_t = \Delta'_t$, and if the coupling is reciprocated then also $\delta_t = \Delta_t$. This case is the most involved, and to ensure that the task graph becomes acyclic, it is important to reiterate that the interact coupling template specifies that the observation made by $\mathbf{O_i}$ at a certain time t, will only be used by the first solving step $\mathbf{S}$ with time $t' > t$. However, then multiple events in B may occur before the next event of A. If the ratio between A and B of events occurred is static, then a time filter may be used on the conduit, which can aggregate events in one direction and interpolate between events in the other as shown in Appendix A. This time filter must know the time step of the receiving submodel for this to behave correctly.

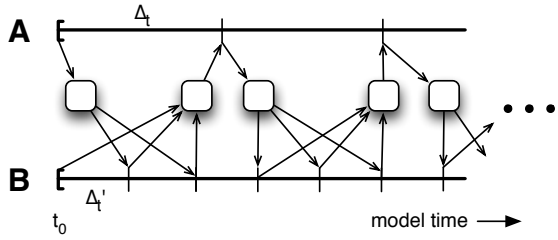
So, a message is never sent backwards in time between pairs of submodels. For



(a) Scale separation



(b) Scale contiguity



(c) Scale overlap

Figure 2.11: Timelines of two submodel instances A and B with temporal scales $S(\Delta_t, \Omega_t)$, $S(\Delta_t', \Omega_t')$. The submodels show temporal scale separation, contiguity and overlap, respectively, and they start at time t_0. Initialisation is done at opening braces ($\mathbf{f}_{\text{init}}$ and $\mathbf{O}_\text{i}$), the last solving step and finalisation at closing braces ($\mathbf{S}$, $\mathbf{B}$, and $\mathbf{O}_\text{f}$), and each intermediate step at a tick ($\mathbf{S}$, $\mathbf{B}$, and $\mathbf{O}_\text{i}$). Arrows indicate messages sent, rounded rectangles denote temporal scale conduit filters aggregating or interpolating between messages.

more submodels, the problem remains submodels with overlapping temporal scales with a varying granularity, which can not be the receiver of the interact coupling template. Indeed, this remains true for indirect interactions, such as when submodel A uses the call coupling template for B, which then has a release template to C. As long

as there is no path of coupling templates starting at $\mathbf{O_i}$ of one submodel and ending at $\mathbf{S}$ of another submodel with a varying granularity, this problem does not appear. Likewise, when A uses the call coupling template for B, which uses the dispatch template for C, which uses the release template for A, the combined total time of B and C might be more than the granularity of A. This problem can only be prevented by a modeller, and it might point to a larger problem with the model: if the combined time of B and C is more than the granularity of A, it might well be that the computed results of B and C are not valid within the timeframe of one step of A. By constructing the task graph, the task graph will detect a cycle and prevent the mistake from happening.

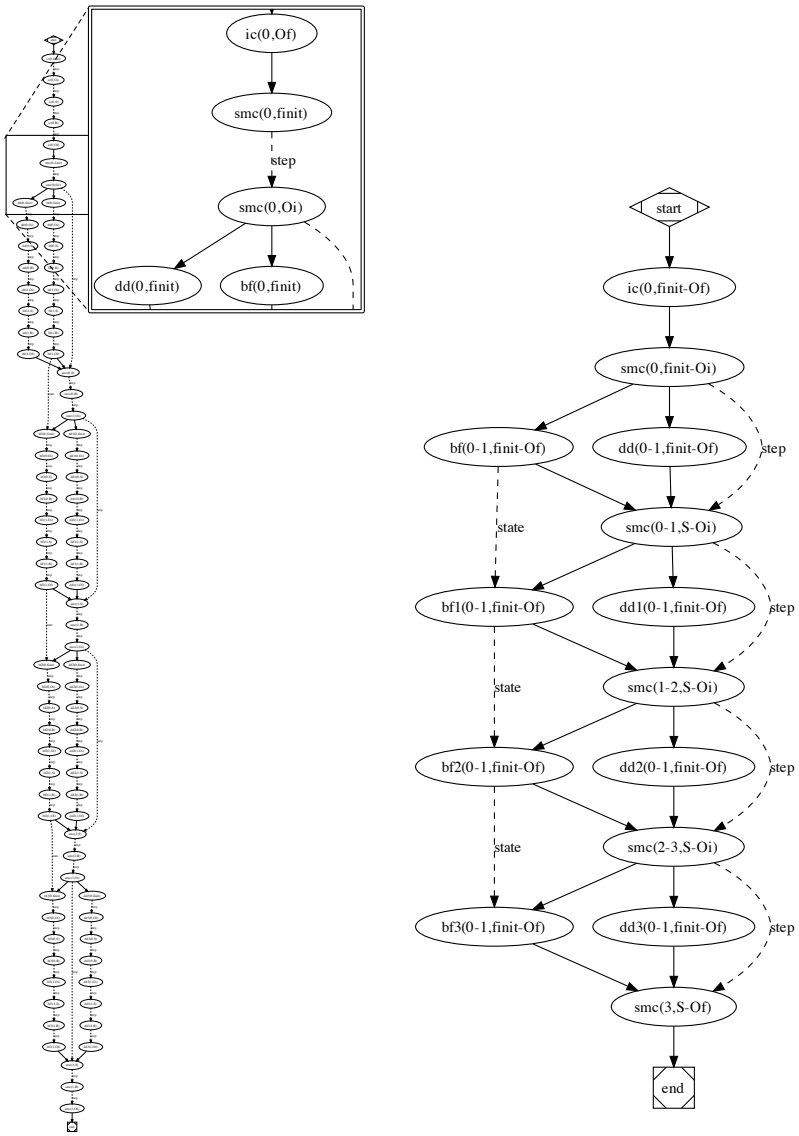
Mappers, not considered so far, can not introduce cycles that were not present in the description above because of their behaviour of sending a message when a message is received.

2.4.3 Task graph reduction

Task graphs can get extremely large, growing exponentially in the number of temporal scale separated submodels, exacerbated by submodels that have a lot of iterations. A methods to reduce the number of nodes is collapsing redundant nodes as in Figure 2.12b. This reduction can be performed while the task graph is constructed or after it is finished. The reduction is straightforward, and can be performed no matter how dynamic the multiscale model is.

Nodes of the task graph can be collapsed only if the graph is guaranteed to remain acyclic. For example, a submodel instance A that only has a input port at $\mathbf{f}_{\text{init}}$ and an output port at $\mathbf{O_f}$ could list all its SEL operators for each iteration as separate nodes, but that would not add any information regarding its dependencies on other submodels. Instead, A can be collapsed to a single node in the task graph, while retaining its dependencies of initialisation and final observation. To mark it as a collapsed node, a range is assigned to the iteration number and SEL operator in the label. The label of a collapsed node encompassing iterations i_0 to i_1 and SEL operator o_0 to o_1 then becomes $\mathit{instanceName}(i_0{-}i_1, o_0{-}o_1)$. In general, node p can be collapsed with node q, forming node r, if the following conditions hold:

- p and q represent the same submodel instance, initialisation number and set number;



(a) Task graph **(b)** Collapsed nodes

Figure 2.12: A task graph of a simplified ISR3D model, which is described in Chapter 4.4, before and after the reduction described in the text. The actual label text is not important in this example, rather the structure of the task graph can be seen to remain intact.

- p has an edge to q; and
- p has exactly one outgoing edge *or* q has exactly one incoming edge.

If this is the case, p and q can be removed and replaced by r, which will have the edges of p and q, while excluding the edge from p to q.

Above conditions ensure that dependencies between submodels remain visible, and that no cycle is introduced in the task graph, as we will show now. Given an acyclic task graph and a reduction from nodes p and q, and suppose a cycle is introduced in the reduction, so that besides a path from s to t, where s and t are non-equal nodes, there is also a path from t to s. First, replacing p and q with r is equivalent to creating an edge from q to p, while disregarding the cycle between p and q. Since only an edge from q to p is added and this causes a path from t to s to be created, there must be

1. a path from t to q not containing p and
2. a path from p to s not containing q.

Suppose this is the case, and suppose that the condition holds that q has exactly one incoming edge, which is from p, then any path to q must contain p, ruling out item 1. Now, suppose that p has exactly one outgoing edge, then any path from p must also contain q, ruling out item 2. Combined, this contradicts that the given task graph was acyclic. Thus, a reduction performed under listed the conditions yields an acyclic graph.

As long as the conditions above hold, the degree of collapse can vary depending on the intended use of the task graph. If, for example, the exact receiving SEL operator of submodel A in the example above must be known, then submodel A can be collapsed into two parts; one node for $\mathbf{f}_{\text{init}}$: $A(0, \mathbf{f}_{\text{init}})$; and another for the rest: $A(0\text{–}n, \mathbf{O}_\mathbf{i}\text{–}\mathbf{O}_\mathbf{f})$. The above list of conditions is then appended with

- p does not have incoming edges from a node of another submodel instance, initialisation number or set number and
- q has exactly one incoming edge.

If the exact sending SEL operator is also of importance, then in the example $\mathbf{O}_\mathbf{f}$ must also be kept separate: $A(n, \mathbf{O}_\mathbf{f})$; ensured by appending the conditions with

- p has exactly one outgoing edge and
- q does not have outgoing edges to a node of another submodel instance, initialisation number or set number.

With these simple guidelines, it is possible to reduce the number of nodes in the task graph of a multiscale model.

2.5 Distributed multiscale execution

A multiscale model, once specified with MML, can be executed in a multitude of ways such as a workflow system, the GridSpace Experiment Workbench 2 [39], or a specialised framework. Because of the modular setup we propose for multiscale models, it is feasible to distribute the computation among multiple heterogeneous machines. For example, a setup with a single laptop possible, but also an execution over a university network or full use of networked supercomputers.

Acyclic parts of a coupling topology will only execute once, and executing them as part of a workflow takes full advantage of existing grid technology. Each node of the compacted task graph can separately be scheduled, choosing the appropriate machine for that task. Output data then needs to be collected at the end of each job. However, cyclic parts of the coupling topology run in this manner will cause delays, since it produces many task graph nodes and each node has a probability of waiting in queue. An alternative is reserving all resources for the duration of a simulation and running each task on an appropriate machine. This requires an active runtime environment to start tasks on those machines and to perform communication. These methods can also be combined: in a coupling topology consisting of a number of cyclic subgraphs that are coupled in an acyclic way, a workflow system can start each cyclic subgraph as a job managed by an active runtime system.

The Multiscale Coupling Library and Environment 2 (MUSCLE 2), described in Chapter 3, is a distributed runtime environment for multiscale models with a cyclic coupling topology. It will start a computational element on a machine when requested and will shut the element down when it indicates that it is finished. MUSCLE 2 uses decentralised message passing, so to assure that that communication data does not need to be stored to file or at a central server, it requires elements that will communicate to be simultaneously active.

The GridSpace Experiment Workbench 2 can start MUSCLE 2 for cyclic coupling topologies, and it can instruct the QCG middleware [84] to start computational elements on several machines simultaneously. More detail on this process is given in Chapter 3.

In this thesis, we do not consider the specifics of scheduling the different parts of a multiscale model to a distributed infrastructure. Rather, we refer to existing task graph literature that addresses this [12, 33, 45, 87].

2.6 Applying the multiscale concepts

The methodology outlined in this chapter is being applied to several multiscale applications in the MAPPER project [144] from five scientific communities: biomedical physics, nano materials, hydrology, fusion and systems biology. By way of example, we will highlight a multiscale model of the formation of clay-polymer nanocomposite materials [132]. The description of a multiscale model of in-stent restenosis is found in Chapter 4.4.

Clay-polymer nanocomposite materials can, in certain formations, be fire-retardant or extremely sturdy. The multiscale model of the formation of these materials evaluates quantum mechanics (QM) for the placement of individual atoms between multiple clay sheets, fine-grained molecular dynamics (FGMD) for the placement of individual molecules, and course-grained molecular dynamics (CGMD) for the placement of groups of molecules. The simulation starts by evaluation QM properties, then the computed values are used in the FGMD submodel, and finally the placement of the molecules is used in the CGMD submodel. Domain decomposition is performed on the QM and FGMD submodels, the first into 6 instances and the second to between 10 and 20 instances.

The three submodels have a typical macro-micro coupling, from QM to FGMD to CGMD, with spatial and temporal scale separation, and a scale ranging from picometer to micrometer and from picosecond to millisecond. Smaller submodels are in a subdomain of larger submodels, so all couplings are sD.

The gMML of Nano materials is shown in Figure 2.13, and also features mappers to manage information coming from the submodels with decomposition. The coupling topology is acyclic, as can be seen from the gMML, so it also has a fixed number of synchronisation points. Also the number of submodel instances is fixed before the

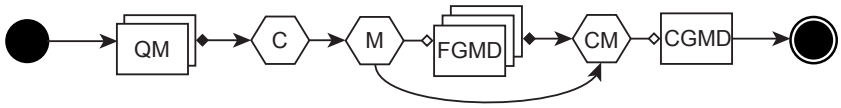


Figure 2.13: The gMML of the Nano materials application. It starts with a few QM submodels, after which their output is collected by the C fan-in mapper. The output is then preprocessed by fan-out mapper M to send it to a number of FGMD submodels. Their output, and the mapping used by M, is collected by the CM fan-in mapper to be used by the final CGMD submodel.

model starts.

The model is implemented using multiple libraries: the LAMMPS and CASTEP, with Perl scripts acting as mappers. The workflow is straightforward to implement. It can be run in a distributed environment using GridSpace 2 as a workflow manager and the Application Hosting Environment (AHE) [154] as a middleware stack.

2.7 Conclusions and further work

Multiscale modelling is being embraced as a paradigm to study and better understand nature. Models produced in this way can be computationally expensive, leading the way to distributed multiscale computing. In this chapter, we have laid foundations for distributed multiscale computing, from a formal background to a full computational specification, which can be analysed to execute on distributed infrastructure. Moreover, using this methodology it is already possible to do distributed multiscale computing.

The formal part has been inspired largely by Complex Automata theory [67] by generalising it to multiscale modelling, and is duly influenced by the Heterogeneous Multiscale Method (HMM) [42], Ingram et al. [76] and the multiscale ontology by Yang and Marquardt [151]. By formally defining concepts as phenomenon, scale and scale separation, we were able to restrict and classify the flow of a submodel and the interaction between submodels. Moreover, the definitions form a basis for decomposing and thus modularising a multiscale model.

The frameworks classification by Ingram et al. [76] is the following: multidomain, embedded, parallel, serial, and simultaneous. The first, multidomain integration consists of a macro-scale model M coupled to a micro-scale model μ on a separate domain. In our methodology this is represented by a mD coupling between μ and M, and sup-

pose these models have temporal scale overlap, then the interact coupling template with the SEL **B** operator will be used. If there are multiple micro-models, their data can first be aggregated by a mapper before M processes it. In an embedded integration, μ models a subdomain of M, which is a sD interaction. The call and release coupling templates could be used here if there is no temporal scale overlap. With parallel integration, two submodels 1 and 2 model the same domain using a different representation. They will have to have a cyclic coupling topology and have temporal scale overlap, and will use the interact coupling templates. With serial integration, one submodel is executed after the other. This is exactly what the dispatch coupling template, and the acyclic coupling topology is meant for. Finally, with simultaneous integration all submodels run at a micro-scale and information is integrated after they run. In our methodology this can be achieved by computing the micro-scale models and sending the output to a fan-in mapper, which can process and aggregate the information computed. The strength of our methodology is that it allows multiple of these strategies to be combined, having for instance parallel integration on one part and multidomain integration on another.

Finally, the multiscale ontology by Yang and Marquardt [151] provides a formal background to multiscale modelling. Unfortunately, although their definitions of domain are very thorough, their definition of scale is too hierarchical to be suitable for multiple dimensions. By redefining scale in this paper, we have a clear way to identify the scale of a submodel and relate it to the scales of other submodels. Their definition of a coupling is similar to ours, although we do not impose laws on the couplings. Instead, we offer conduit filters that modellers can use to enforce a rule on a conduit, which is a more low-level way to represent a coupling laws. Interesting definitions that we do not reproduce are those of couplings between elements or subdomains of two submodels and composing submodels to form a system that again acts as a submodel.

From the restricted behaviour of our formalism, a computational specification language, the Multiscale Modelling Language (MML) [47], is refined and firmly based. It takes advantage of scales but also includes well-defined computational elements such as one-way conduits, conduit filters and mappers, used to interconnect the different submodels. It is a high-level language, leaving the implementation of those elements free. Additionally, since it is constrained by the definitions in multiscale modelling, it is more specific and concrete than other component-based modelling specifications, such as CCA [8].

Given such a focused multiscale model specification with well-defined elements, its properties can be deduced: how dynamic it is, what dependencies and possibilities for deadlock appear between submodels, and how the model could be scheduled over multiple resources. We propose a method to represent the execution of a multiscale model with a task graph, which can in turn be used to schedule submodels over multiple (distributed) resources [33, 45]. Additionally, if all computational elements adhere to the MML specification, possible deadlocks can be detected before execution while constructing the task graph.

Finally, a properly defined and analysed multiscale model can be executed for example by the Multiscale Coupling Library and Environment (MUSCLE) [65]. Within MUSCLE, conduits and some filters are already supplied and submodels and mappers can be implemented in Java or C++, or any programming language that has an interface with C++, such as Fortran or C. MUSCLE is under active development, which will be the basis for a separate contribution. MUSCLE does not do scheduling, so for distributed multiscale computing a combination with grid middleware such as QosCosGrid tools [84] is being explored.

Alternatives to MUSCLE for executing a multiscale model are possible, seeing that MML is not tied to a specific implementation. Particularly, if a multiscale model has a acyclic coupling topology then there are much simpler systems than to execute it. In such a case, a lot of workflow composition tools would also suffice, of which multiple are quite suited for distributed execution [7, 30].

One of the goals of the MAPPER project is to further automate the combined workflow of specifying, analysing, and doing distributed execution of a multiscale model by developing additional MML tools and integrating them with QosCosGrid and GridSpace 2 [39]. This will be described in Chapter 3.

This chapter addresses *how* to approach distributed multiscale computing, in the next chapters we will see it implemented and focus on the *performance* of doing distributed multiscale computing as opposed to local computing.

In this thesis no solution is offered for models with cyclic coupling topologies that contain submodels with non-regular temporal scales. First of all, if two submodels with temporal scale specifications S and S' (with $\Omega > \Omega' \vee (\Omega = \Omega' \wedge \Delta \geq \Delta')$) have no no overlap, separation, or contiguity, i.e.,

$$(\Delta \geq \omega' \vee \Delta' \geq \omega) \wedge \Omega' \geq \delta \wedge (\Delta' > \delta \vee \Omega' > \Delta),$$

they will not abide by the defined coupling templates. If they have scale overlap and more than two submodels with non-regular temporal scales are interacting, advanced synchronisation schemes must be employed [54, 77], which are not covered by the theory in this thesis. If two submodels have scale separation, there is no problem since the coarse scale submodel is not exposed to event-based state changes, and the fine scale model is restarted for each event. Allowing non-regular temporal scales in cyclic task graphs may introduce non-determinism for the dependencies in the task graph, and the full implications will need further study.